Parallel Implementation of a Genetic Algorithm for the Travelling Salesman Problem

Bianca Ziliotto

2 February 2024

Abstract

This work explores possible parallelization patterns for a genetic algorithm applied to the Travelling Salesman Problem. After evaluating the theoretical speed-up of multiple alternative solutions, the candidate solution is implemented both using native C++ threads and FastFlow. An experimental analysis is finally provided to validate the theoretical results.

1 Introduction

1.1 Travelling Salesman Problem (TSP)

The TSP is a well known NP-hard optimization problem, defined as follows: given a set of cities $\{i\}_{i=1}^{n}$ and the matrix $\mathcal{D} \in \mathbb{R}^{n\times n}$ of pairwise distances between them, find the shortest possible route $\mathcal{R} \in \mathbb{N}^{n-1}$ that visits every city exactly once and returns to the starting point. (Notice that since the route is cyclic, we can consider city 1 as the starting and ending point, with no loss of generality. Hence, a route $\mathcal{R} \in \mathbb{N}^{n-1}$ is identified by any permutation of the remaining $n-1$ cities).

1.2 Genetic algorithm

A genetic algorithm is a heuristic search algorithm that is often used for optimization problems like TSP, unfolding in the following phases (Figure 1):

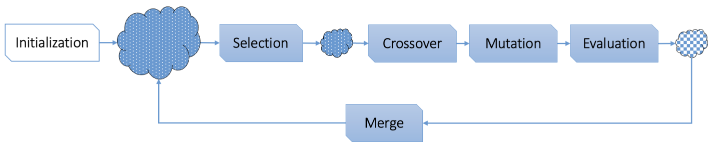


Figure 1: Genetic algorithm

- Initialization: the initial population is generated and the fitness score is computed for each individual.
- At each iteration, a new population is derived from the previous one, up to a maximum number of generations, through the following steps:
  - Selection: a smaller set of individuals is sampled from the original population, with a probability based on their fitness score.
  - Evolution: crossover and mutation are applied to selected individuals.
  - Evaluation: new scores are computed for evolved individuals.
  - Merge: a new population is created with the fittest individuals from the old and the new population. (Also, the new probabilities for selection are computed).

1.3 Dataset

The dataset that we will use throughout the experiments is `it16862`([1]), which contains 16,862 locations in Italy and is derived from the National Imagery and Mapping Agency data.

1.4 Sequential time

Before moving to the evaluation of alternative parallelization patterns, we are interested in the sequential time required by each phase of the algorithm, in such a way to identify the most expensive steps. We execute the algorithm for two different sizes of the population, measuring the initialization time and computing the average time of the other phases over 20 iterations.

Phase	**Time [μs] (500)**	**Time [μs] (5000)**
Initialization	8.432×10^5	3.068×10^6
Selection	$(2.515 \pm 1.243) \times 10^3$	$(2.674 \pm 1.220) \times 10^4$
Crossover	$(1.375 \pm 0.771) \times 10^7$	$(1.387 \pm 0.019) \times 10^8$
Mutation	$(0.805 \pm 0.846) \times 10^1$	$(1.057 \pm 1.050) \times 10^2$
Evaluation	$(1.111 \pm 0.014) \times 10^5$	$(1.118 \pm 0.004) \times 10^6$
Merge	$(1.228 \pm 0.392) \times 10^4$	$(7.846 \pm 3.674) \times 10^4$
Total	2.783×10^8	2.802×10^9

Table 1: Cost of different steps in sequential algorithm.

Despite the high variance of some phases, we can undoubtedly identify the crossover as the most expensive step of each iteration. We also notice that all times are proportional to the size of the population, which is not surprising considering that the same operations are executed on each individual.

2 Theoretical Analysis

Since every iteration starts from a population of individuals, we can derive multiple sub-tasks from the same input task, rather than from an input stream. Thus, the problem suggests to exploit the advantages of **data parallelism**.

The first question that we address is **how frequently** different workers need to realign together. Defining a lower bound is straightforward: since merging operation requires sorting the entire population by fitness, it is necessary to join threads at least once per iteration. It is immediate to conclude that such joining should happen exactly once per iteration, as doing it more frequently would result in a useless increase of overheads.

The second choice consists in deciding **when** threads should be forked and joined, i.e. which phases should be executed in parallel (Figure 2). We decide not to parallelize the initialization, as its cost is fully amortized when the number of iterations grows (genetic algorithms typically require thousands of iterations). Therefore, we only parallelize phases that are repeated at each iteration.

Selection can be executed in parallel, provided that the vector of cumulative probabilities has already been computed on the entire population. In particular, it is possible to directly work on chunks of the selected population, while sampling a subset of individuals from the entire original population. Crossover, mutation and evaluation are also completely parallelizable, thus will be executed concurrently on different chunks of the selected population.

As for the merge operation, we need to concatenate different chunks (previously evaluated), sort them by fitness, and truncate the population to the original population size, before moving to a new selection. Since evaluation has already been parallelized, and parallelizing concatenation would be of little benefit while introducing additional overheads due to necessary synchronization mechanisms, we decide to let this section fully sequential. Furthermore, we compute the total fitness and the vector of cumulative probabilities allowing the following selection to be executed completely in parallel. The sequential work time t_{seq} can then be split into a serial fraction t_s and a non-serial fraction t_{ns}:

$$t_{seq} = t_s + t_{ns}, \text{ where: } \begin{cases} t_s = t_{init} + n_{gen} \times t_{merge} \\ t_{ns} = n_{gen} \times (t_{sel} + t_{evol} + t_{eval}) \end{cases}$$

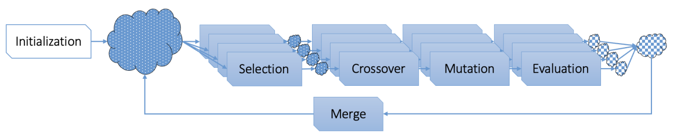


Figure 2: Parallelization of the genetic algorithm.

The third issue concerns designing the **modality** in which tasks are assigned to workers, and consequently defining the dimension of such tasks. A possibility is splitting the selected population into chunks and statically assign each chunk to a worker. Alternatively, we can opt for a dynamic assignment, by creating a queue of minimal tasks and letting each worker execute an arbitrary number of tasks, in such a way to better balance the workload.

Static assignment. If we divide the input data into n_w chunks and let each of the n_w threads work on each chunk, we get:

$$t^S_{n_w} = t_s + t^S_{ns} + t^S_{overhead}(n_w) = t_s + \max_{i=\{1,\ldots,n_w\}} t_i + t^S_{overhead}(n_w)$$

If the workload is balanced, we can assume $\max_{i=\{1,\ldots,n_w\}} t_i \approx \frac{t_{ns}}{nw}$.

Dynamic assignment. If we opt for a dynamic assignment, we define a minimal task as a chunk of 2 individuals (in such a way to allow a crossover operation within a single task).

$$t^D_{n_w} = t_s + t^D_{ns} + t^D_{overhead}(n_w) \approx t_s + \frac{t_{nc}}{n_w} + t^D_{overhead}(n_w)$$

While a better balancing is guaranteed by the dynamic pattern, improving t^D_{ns} over t^S_{ns}, a higher effort is required to synchronize the frequent accesses to the queue of tasks, increasing the impact of $t^D_{overhead}(n_w)$. The impact of all these terms is further investigated in the following sections.

2.1 Ideal speed-up

If we had implemented every operation in parallel, the ideal speed-up would be $sp(n_w) = n_w$. However, having left some operations sequential, due to necessity or opportunity, a stricter upper bound on the expected speed-up is given by the **Amdhal law**:

$$\text{speed-up}(n_w) = \frac{1}{f + \frac{(1-f)}{n_w}}$$

where f is the **serial fraction** t_s/t_{seq}. Based on results collected in Section 1.4, we reconstruct the values of t_s and f for 20 iterations, thus derive an estimation of the ideal speed-up (Table 2):

	500 individuals	**5000 individuals**
t_s	$1.089 \times 10^6 \mu s$	$4.637 \times 10^6 \mu s$
f	0.39%	0.16%

Table 2

Since we have no reason to believe that the serial fraction should vary with the size of the population, we assume that the difference is due to the intrinsic stochasticity of the algorithm and we consider the mean value **0.28 %** as the expected serial fraction for both population.

2.2 *A priori* analysis of non-functional properties

The factors that could negatively affect the speed-up of the parallel implementation are the workload imbalance and the overheads, including the time spent to fork and join threads and the time spent to split tasks and gather results. Quantifying the impact of these factors *a priori* is not trivial; in this section, we will attempt to estimate their order of magnitude, before verifying it with direct measurements in the experimental phase.

Concerning the **workload imbalance** in static assignment, we can refer to the standard deviations collected in Section 1.4. Limiting ourselves to consider crossover (which is the most expensive stage of the algorithm), for the populations considered we have a standard deviation in the order of $10^6\mu s$. Such deviation can be considered an indicator of the source of imbalance, and will of course increase as the dimension (and cost) of sub-tasks increases. For dynamic assignment, the waiting time for a thread is bounded by the cost of a single minimal task (made of 2 individuals), which is in the order of $10^5\mu s$.

As for the **overheads**, we know that forking and joining threads takes from hundreds of μs to few ms (depending on the number of threads). For this reason, at least in the static assignment, we expect overheads to have a smaller impact than workload imbalance. As for the dynamic assignment, we expect an additional overhead for the construction of the queue of tasks and for the synchronization of the thread accesses.

3 Implementation

The parallel implementation of the algorithm is provided both using native C++ threads (Section 3.1) and using the Fastflow library (Section 3.2).

3.1 Native C++ threads

Since the time spent for crossover operation is at the same time the one with highest mean and highest variance, we consider the benefits of dynamic assignment to be worth exploring: for this reason, we decide to provide both static and dynamic implementations. Both implementations use `std::thread`, and parallelize the same phases of the algorithm.

For the **static** implementation (`tsp_static.cpp`), we simply split the vector of the selected population into chunks, assigning each chunk to a thread. Threads are forked after the probabilities for selection have been computed, and joined after the evaluation before the merging operation. No synchronization mechanism is needed, thus overheads are reduced do the minimum. For this reason, provided that the population is large enough to make parallelization worth, the main factor that we expect to weaken the performance of the static implementation with respect to the dynamic assignment is the possible workload imbalance.

For the **dynamic** implementation (`tsp_dynamic.cpp`), we split the work into minimal tasks, where each task consists in performing selection, crossover, mutation and evaluation on a couple of individuals (as at least two individuals are needed for crossover). Tasks are stored in a queue and each worker repeatedly pops and solves tasks from the queue, until the queue is empty. In this case, a `std::mutex` is used to manage concurrent accesses to the queue, preventing races.

3.2 Fastflow

An alternative version (`tsp_fastflow.cpp`) is implemented using the Fastflow library: in particular, we choose to exploit the `poolEvolution` pattern, expressly designed for genetic algorithms.

Apart from the initial population, the `poolEvolution` pattern requires the following routines as parameters: `selection` (selection), `evolution` (crossover, mutation, evaluation), `filter` (merge), `termination` (checking the stopping criterion); eventual variables needed by the algorithm are stored in a struct named `Env_t`. The only peculiarity that we must take into account is that `evolution` is executed concurrently on single individuals. This only introduces a slight difference in the implementation of crossover: while in the previous implementation two parents generate two children, here we only have one parent generating one children. The solution consists in sampling the second parent from the selected population, stored in `Env_t`, and finally generating an only child. This might result in slight performance differences, both in terms of convergence and computational cost.

4 Experiments

In this section, we compare the different implementations in terms of multiple performance metrics (speed-up, scalability, efficiency). Then, we explore the sources of imbalance and overheads, to confirm and eventually correct the assumptions made in Section 2.2.

4.1 Experimental set-up

Experiments are run on a machine with 32 physical cores, evenly distributed among 2 sockets, each core supporting 2 threads. We test our parallel implementations while varying the number of threads; in particular, we execute each algorithm with $n_w = 1, 2, 4, 8, 16, 32$ and finally with $n_w = 64$ (hyper-threading).

All experiments are repeated for the two populations (of 500 and 5000 individuals). During each experiment, the genetic algorithm is executed on the same dataset, for a fixed number of generations (10), with the same parameters (selection ratio = 50%, mutation ratio = 50%).

The sequential version (`tsp_sequential.cpp`) of the algorithm, executed within the exact same settings, provides the reference for the performance metrics.

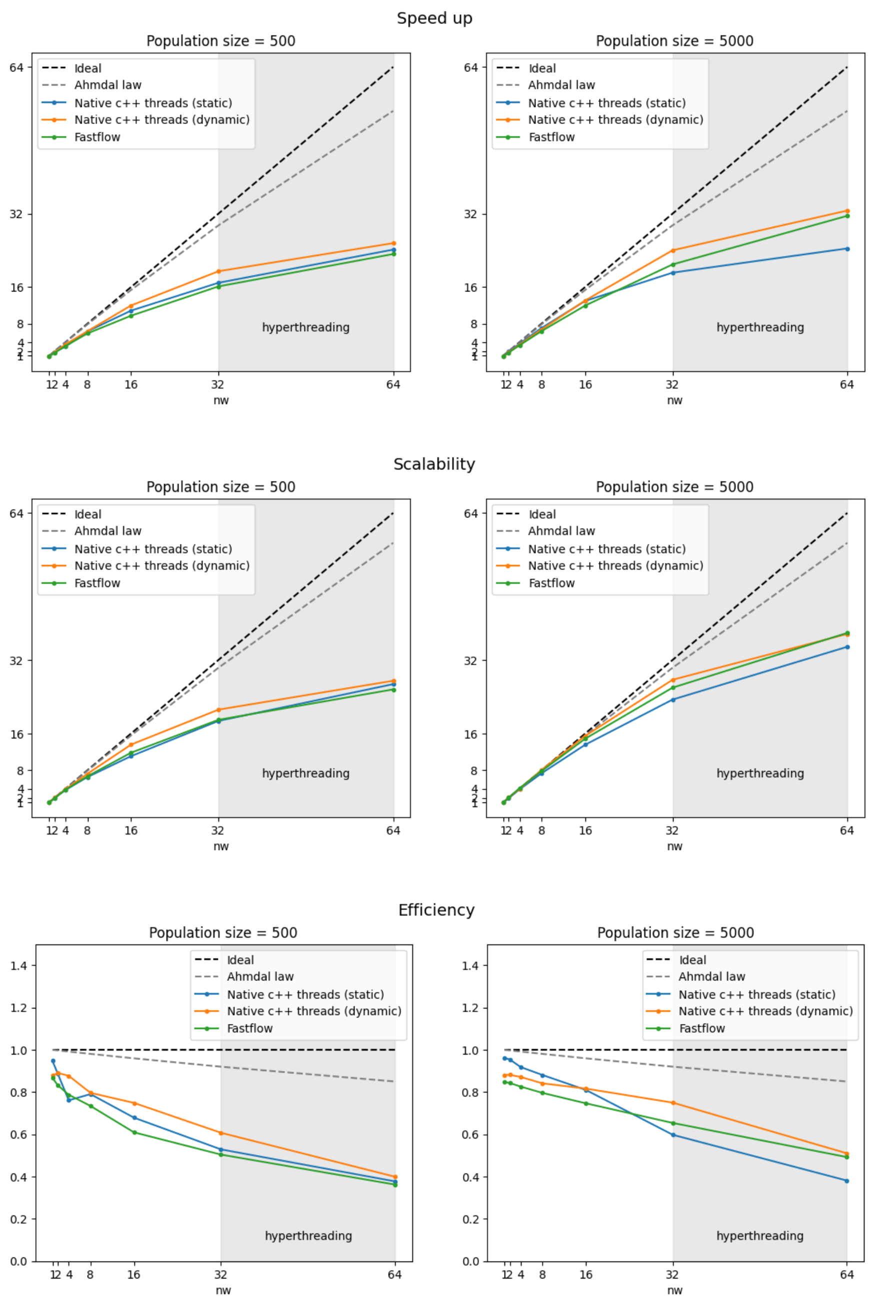


Figure 3: Experimental results.

4.2 Discussion of results

Figure 3 shows the speed-up, scalability and efficiency obtained with native C++ threads and Fastflow on the two different-sized populations.

Each plot shows the actual performance obtained with the three algorithms, compared to the ideal case (complete parallelization, perfect balancing, absence of overhead) and to the ideal case considering the Ahmdal law (taking into account our estimate of the serial fraction, always assuming perfect balancing and absence of overheads).

As expected, for few number of threads the three algorithms have a performance close to the ideal case; as the parallelization degree increases, overheads have a progressively higher impact and the exploitation of resources ceases to be optimal. In agreement with theoretical predictions, performance are significantly better on the larger population, as the impact of overheads is better absorbed by the expensive workload.

In every plot, we notice that dynamic implementation outperforms the static pattern both in terms of speed-up and scalability: this is particularly stressed in the case of the larger population, once again due to the relatively smaller impact of overheads.

As for the results obtained with Fastflow, we observe that `poolEvolution` performs usually better that the static implementation, especially for large populations; this should not surprise us once we have assessed the benefits of dynamic assignment, considering that `poolEvolution` relies on the same paradigm.

Looking at the efficiency plot, we can see how the static implementation has a higher efficiency for small nw, but then decreases much faster than the two other implementations: this could also be expected considering that if we only use 1 thread the impact of imbalance is null, while the additional overheads of the dynamic implementation are significant.

4.3 *A posteriori* analysis of non-functional properties

Following to our first attempt to theoretically estimate the order of magnitude of non-functional properties (Section 2.2), we conclude our analysis trying to quantify empirically the impact of imbalance and overheads, through a series of direct and indirect measurements.

For this experiment, we restrict to the native C++ threads implementation, and we increase the number of threads up to 32.

Using `utimer.hpp`, we measure the following times at each iteration, and average them over n_{gen}=20 iterations:

- Mean work time per thread $\bar{t}_i$;
- Maximum work time per thread $\max_i t_i$;
- Non-serial fraction time t_{ns}^{iter};
- Serial fraction time t_{ns}^{iter};

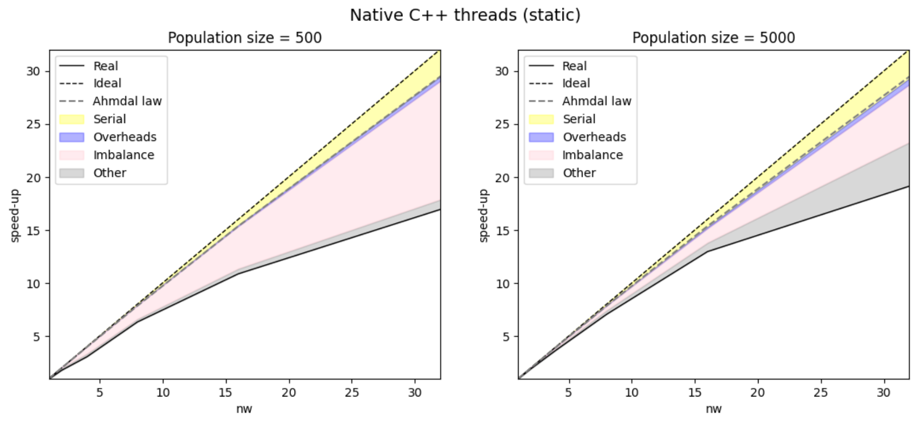
- Total iteration time $t^{iter} = t_s^{iter} + t_{ns}^{iter}$.

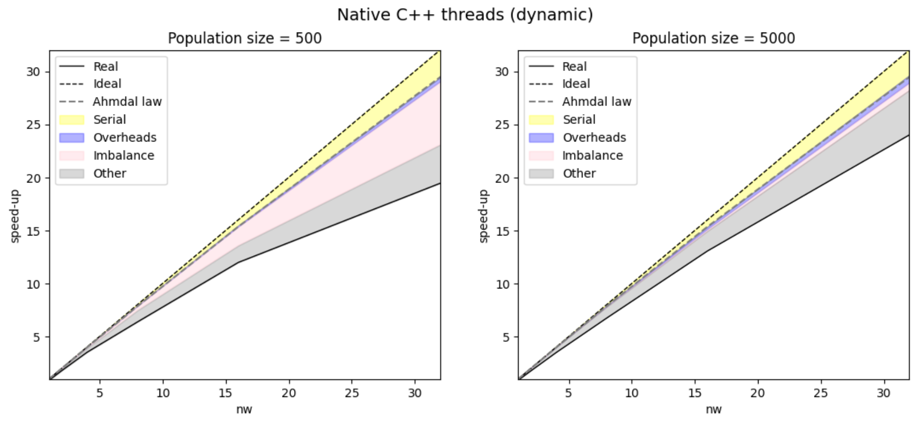
Also, we keep track of the initialization time t_{init} and the overall time required by the algorithm.

From the above measurements, we derive an estimation of the following factors *a posteriori*:

- **Imbalance**, defined as $n_{gen} \times (\max_i t_i - \bar{t}_i)$;
- **Overhead**, defined as $n_{gen} \times (t_{ns}^{iter} - \max_i t_i)$;
- **Serial time**, defined as $t_{init} + n_{gen} \times t_s^{iter}$.

At the light of these measurements, we give a visual representation how how different factors limit the speed-up of the static and dynamic implementations.





First of all, we verify the accuracy of our estimation of the serial fraction (2.1), by comparing the expected Ahmdal speed-up with the speed-up that we would reach taking into account the real serial time.

Concerning load balancing, we verify that the auto-scheduling approach drastically reduces the imbalance, especially for larger populations.

As for the overheads introduced by parallelization, we partially verify what we predicted in 2.2: the impact of overheads is negligible with respect to the one of imbalance in the static implementation, while the two factors become comparable in the dynamic assignment. Also, we observe that not only the overheads increase proportionally to the number of workers (as expected), but also to the size of the population (which is probably due to the cost of splitting the input task and concatenating all partial results).

Finally, we notice that a portion of the region between ideal and real speed-up lines is not attributable to any of the previous factors (serial fraction, imbalance, overheads). Possible explanations include additional overheads introduced by time measurements, intrinsic stochasticity in the algorithms, fluctuations in resources allocation, memory overheads. Furthermore, it is worth noting that the main thread is only waiting for the other threads to execute sub-tasks, thus in every experiment there is an extra thread that is not working, making the number of threads $nw + 1$: this implies that when $nw = 32$, we are already in a hyper-threading regime (as we are working on a 32-cores machine). This last aspect could also lead to a worsening of performance with respect to the ideal the speed-up.

References

[1] it16862. Available at: `https://www.math.uwaterloo.ca/tsp/world/`.